Лабораторная работа 3.

Черновик 0.6

Целью лабораторной работы является знакомство студентов с обработкой текстовых файлов.

Студенты должны получить и закрепить на практике следующие знания и умения:

1. Обрабатывать текстовые файлы (открывать в различных режимах, читать и записывать информацию).
2. Организовывать корректную работу с ресурсами (в данном случае – файловыми описателями).
3. Анализировать информацию об ошибках с помощью функций стандартной библиотеки.
4. Использовать в программе аргументы командной строки.

1. Общее задание

1. Исходный код лабораторной работы располагается в отдельной ветке lab_03. В ветке lab_03 для каждой задачи создается папка lab_03_X_Y, где X – номер задачи, Y – номер варианта (например, если для первой задачи вы решаете 3 вариант, папка будет называться lab_03_1_3).
2. Исходный код должен соответствовать правилам оформления исходного кода.
3. Для каждой задачи создается отдельный проект в *QT Creator*. Для каждого проекта должно быть два варианта сборки: Debug (с отладочной информацией) и Release (без отладочной информации).
4. Для каждой задачи студентом подготавливаются тестовые данные, которые демонстрируют правильность ее работы. Входные данные должны располагаться в файлах in_z.txt, выходные out_z.txt, где z – номер тестового случая. Тестовые данные готовятся и помещаются под версионный контроль еще до того, как появится реализация задачи.
5. Для реализации любой из задач этой лабораторной работы вам необходимо выделить, по крайней мере, одну осмысленную функцию. Необходимо предусмотреть обработку ошибочных ситуаций.

2. Индивидуальное задание

`Номер задания = Номер в журнале % Количество вариантов.`

Задача 1

Пользователь вводит целые числа, по окончании ввода чисел нажимает Ctrl-Z и Enter или вводит букву.

Написать программу, которая

0. находит наибольшее положительное из чисел, которые следуют за отрицательным числом;
1. находит два максимальных элемента последовательности (возможно совпадающих);
2. находит порядковый номер максимального из чисел (если чисел с максимальным значением несколько, то должен быть найден номер первого из них);

3. определяет сколько раз в последовательности чисел меняется знак (нуль считается положительным числом);
4. находит количество чисел, которые больше своих «соседей», т.е. предшествующего и последующего;
5. находит наибольшее число подряд идущих элементов последовательности, которые равны друг другу;
6. находит наибольшую длину монотонного фрагмента последовательности (то есть такого фрагмента, где все элементы либо больше предыдущего, либо меньше);
7. определяет количество локальных максимумов в последовательности (Элемент последовательности называется локальным максимумом, если он строго больше предыдущего и последующего элемента последовательности. Первый и последний элемент последовательности не являются локальными максимумами.);
8. определяет наименьшее расстояние между двумя локальными максимумами последовательности.

Требования к решению задачи:

1. Прототип функции, которая реализует решение задачи, должен выглядеть следующим образом:

```
int process(FILE *f [, прочие выходные параметры]);
```

2. Функция process возвращает 0 в случае успешного решения задачи и отрицательный код ошибки в противном случае (например, -1 – входных данных нет и т.д.). Для каждого кода ошибки задается мнемоническое имя с помощью директивы define.
3. При решении любой задачи из варианта 1 два цикла ввода и массивы не использовать.
4. Необходимо подготовить наборы тестовых данных по классам эквивалентности. Каждый набор разместить в текстовом файле.

Задача 2

Написать программу, которая считывает из текстового файла вещественные числа и выполняет над ними некоторые вычисления:

0. найти число, наиболее близкое к среднему значению всех чисел;
1. найти количество чисел, значение которых больше среднего арифметического минимального и максимального чисел;
2. рассчитать дисперсию чисел (математическое ожидание и дисперсия рассчитываются отдельно);
3. проверить выполняется ли правило «трех сигм» для чисел;
4. найти среднее значение чисел, расположенных между минимальным и максимальным числами («между» - не по значению, а по расположению).

Требования к решению задачи:

1. При решении задачи выделить несколько функций.
2. При решении задачи массивы не использовать.
3. Имя файла берется из аргументов командной строки.
4. Предусмотреть обработку ошибок.
5. Решение любой из этих задач выполняется минимум за два просмотра файла.
6. Подготовить тестовые данные, демонстрирующие правильную работу программы.

Задача 3

Написать программу, которая обрабатывает двоичный файл, содержащий целые числа. Программа должна уметь

- создавать файл и заполнять его случайными числами;
- выводить числа из файла на экран;
- упорядочивать числа в файле.

Прежде чем реализовывать функцию упорядочивания файла, необходимо реализовать функцию *get_number_by_pos*, которая по заданной позиции, позволяет прочитать число в указанной позиции, и функцию *put_number_by_pos*, которая позволяет записать число в указанную позицию. Функцию упорядочивания необходимо реализовать с помощью этих функций.

В начале файла, содержащего исходный код программы, должен располагаться многострочный комментарий, в котором необходимо указать детали реализации этой задачи: как минимум, выбранные целочисленный тип, алгоритм сортировки, «направление» упорядочивания.

3. Работа с текстовыми файлами

Стандартная библиотека ввода/вывода Си – это библиотека буферизированного обмена с файлами и устройствами, которые поддерживает операционная система. К таким устройствам относятся консоль, клавиатура принтеры и многое другое.

Для взаимодействия программы с файлом или устройством библиотека использует тип **FILE**, который описывается в заголовочном файле stdio.h. При открытии файла или устройства возвращается указатель на объект этого типа (*файловый указатель*).

fopen (параграф 7.19.5.3 стандарта)

```
#include <stdio.h>

FILE* fopen(const char *filename, const char *mode);
```

Функция fopen открывает файл. Она получает два аргумента – строку с именем файла и строку с режимом доступа к файлу. Имя файла может быть как абсолютным, так и относительным. fopen возвращает файловый указатель, с помощью которого далее можно осуществлять доступ к файлу. Если вызов функции fopen прошёл неудачно, то она возвратит NULL.

Некоторые режимы открытия файла.

Режим (mode)	**Описание**
"r"	Чтение. Файл должен существовать.
"w"	Запись. Если файл с таким именем не существует, он будет создан, в противном случае его содержимое будет потеряно.
"a"	Запись в конец файла. Файл создаётся, если не существовал.

Функция fopen может открывать файл в текстовом или бинарном режиме. По умолчанию используется текстовый режим. Если необходимо открыть файл в бинарном режиме, то в конец строки добавляется буква b, например "rb", "wb", "ab".

Замечание

Текстовый файл содержит только печатные символы (т.е. символы которые можно вводить с клавиатуры). Он организован в виде последовательности строк, каждая из которых заканчивается символом новой строки ‘\n’. В конце последней строки этот символ не является обязательным.

Требования операционной системы к символу новой строки в текстовом файле могут быть различными (например, Windows: CR+LF, Linux: LF, Mac: CR). Поэтому чтобы программа была переносимой, символ ‘\n’ преобразуется библиотекой ввода/вывода в представление принятое в конкретной операционной системе.

По этой причине в текстовом файле может не быть однозначного соответствия между символами, которые пишутся (читаются), и теми, которые хранятся в файле.

fclose (параграф 7.19.5.1 стандарта)

```
#include <stdio.h>

int fclose(FILE *f);
```

Функция fclose закрывает файл: выполняет запись буферизированных, но еще незаписанных данных, уничтожает непрочитанные буферизированные входные данные, освобождает все автоматически выделенные буфера, после чего закрывает файл или устройство. Возвращает EOF в случае ошибки и нуль в противном случае.

EOF – макрос, который определен в файле stdio.h. Представляет собой отрицательное целое число типа int, которое используется для обозначения конца файла.

Замечание

Операционная система после завершения программы освобождает все ресурсы, которые программа использовала. Поэтому даже если функция fclose не была вызвана, ничего страшного. Зачем же использовать fclose? Можно назвать несколько причин:

1. У программы может быть ограничение на количество одновременно открытых файлов. Если не закрывать неиспользуемые, то может произойти ошибка открытия очередного файла.
2. В Windows когда программа открывает файл, другие программы не могут его открыть или удалить. Поэтому если файл может кому-то понадобиться и, до окончания работы программы ещё далеко, вызывайте fclose.
3. Вывод в файл совершается не сразу, а через вспомогательный буфер. Это сделано для ускорения процесса вывода. Поэтому при закрытии файла выполняется запись буферизованных, но ещё не записанных данных. Если же fclose не была вызвана, то буферизованные данные могут быть потеряны. Поэтому, если вы хотите гарантированно увидеть всё, что вывела программа, то используйте fclose.

fprintf (форматированный вывод, параграф 7.19.6.1 стандарта)

```
#include <stdio.h>

int fprintf(FILE *f, const char *format, ...);
```

Функция fprintf преобразует и выводит данные в файл (или устройство), связанный с файловой переменной f, под управлением строки форматирования format. Возвращает количество записанных символов или, в случае ошибки – EOF.

fscanf (форматированный ввод, параграф 7.19.6.2 стандарта)

```
#include <stdio.h>

int fscanf(FILE *f, const char *format, ...);
```

Функция fscanf считывает данные из файла (или устройства), связанного с файловой переменной f, под управлением строки форматирования format и присваивает преобразованные значения последующим аргументам.

Функция завершает работу, когда «исчерпывается» строка форматирования. При этом она возвращает EOF, если до преобразования очередного значения ей встретился конец файла или появилась ошибка. В противном случае функция возвращает количество введенных значений.

rewind (параграф 7.19.9.5 стандарта)

```
#include <stdio.h>

void rewind(FILE *f);
```

Функция rewind позволяет начать обработку файла сначала.

Замечание

Согласно стандарту c99 вызов функции rewind может быть заменен следующим вызовом функции fseek

```
(void) fseek(f, 0, SEEK_SET);
```

Функция fseek (в отличие от rewind) возвращает признак ошибки. У нее следующий прототип (параграф 7.19.9.2 стандарта):

```
int fseek(FILE *f, long int offset, int origin);
```

Предопределенные файловые переменные

При инициализации программы библиотека ввода/вывода заводит три файловые переменные *stdin*, *stdout* и *stderr* для:

- Стандартного потока ввода (сокращение от standard input). Программа может читать из него данные.
- Стандартного потока вывода (сокращение от standard output). Программа может выводить в него данные.
- Стандартного потока ошибок (сокращение от standard error). Программа может выводить в него сообщения об ошибках. Это нужно для того, чтобы они не терялись в случае перенаправления stdout.

Обычно стандартные потоки направляются к консоли, но они могут быть перенаправлены операционной системой на другое устройство.

```
# весь вывод программы a.exe на экран будет перенаправлен в файл out.txt
a.exe > out.txt

# ввод данных программы b.exe будет выполняться из файла in.txt
b.exe < in.txt
```

Эти файловые переменные могут использоваться в любой функции, где используется переменная типа FILE*:

```
...

float a, b;

if (scanf("%f%f", &a, &b) != 2)
    printf("I/O error\n");
else

...
```

```
...

float a, b;

if (fscanf(stdin, "%f%f", &a, &b) != 2)
    fprintf(stdout, "I/O error\n");
else

...
```

feof (параграф 7.19.10.2 стандарта)

```
#include <stdio.h>

int feof(FILE *f);
```

Функция feof проверяет наличие установленного признака конца файла. Она возвращает ненулевое значение, если обнаружен установленный признак конца файла, иначе ноль.

Замечание

Поведение функции feof отличается от поведения функции eof языка Pascal. В языке Pascal функция eof возвращает true, если следующая файловая операция (например, read) не выполнится, потому что будет достигнут конец файла. Функция feof возвращает отличное от нуля значение, если последняя файловая операция не была выполнена из-за достижения конца файла.

Пример 1.

Программа получает на вход текстовый файл test.txt, который содержит последовательность из целых чисел. Необходимо определить значение наибольшего элемента этой последовательности.

```
#include <stdio.h>

int get_max(FILE *f, int *max)
{
    int num;

    if (fscanf(f, "%d", max) == 1)
    {
        while (fscanf(f, "%d", &num) == 1)
            if (num > *max)
                *max = num;

        return 0;
    }

    return -1;
}

int main(void)
{
    FILE *f;
```

```
    int max;

    f = fopen("test.txt", "r");
    if (f == NULL)
    {
        printf("I/O error\n");

        return -1;
    }

    if (get_max(f, &max) == 0)
        printf("max is %d\n", max);
    else
        printf("There are not enough data.\n");

    fclose(f);

    return 0;
}
```

Пример 2.

Этот пример должен пояснить почему использование функции feof может приводить к странным результатам при обработке файла. Контроль ошибок в примере исключен, чтобы сократить размер программы.

Программа	**Содержимое test.txt (признак конца строки указан явно специально)**

`#include <stdio.h>`
 	
`int main(void)`
`{`
`  FILE *f = fopen("test.txt", "r");`
`  int num;`
 	
`  while (!feof(f))`
`  {`
`  fscanf(f, "%d", &num);`
`  printf("%d ", num);`
`  }`
 	
`  fclose(f);`
 	
`  return 0;`
`}`
	`1\n`
	`2\n`
	`3\n`
	`4\n`
	`5\n`
	`\n`

Эта программа напечатает число 5 два раза. Почему? После чтения числа 5 функция feof вернет значение 0, и цикл продолжится. Следующий вызов fscanf закончиться неудачно, потому что достигнут конец файла. При этом переменная num содержит значение 5, прочитанное на предыдущей итерации. Это значение и будет еще раз выведено на экран. После этого функция feof вернет значение отличное от нуля, и цикл завершится.

Чтобы исправить эту ошибку, программу можно переписать следующим образом:

```
#include <stdio.h>
```

```
int main(void)
{
    FILE *f = fopen("test.txt", "r");
    int num;

    while (1)
    {
        fscanf(f, "%d", &num);
        if (feof(f))
            break;
        printf("%d ", num);
    }

    fclose(f);

    return 0;
}
```

Однако лучше использовать способ чтения, использованный в примере 1.

4. Параметры командной строки

Материал данного раздела частично приводится по книге С.В. Шапошниковой «Особенности языка Си» [1].

Часто данные передаются в программу из командной строки при ее запуске. Например,

```
# gcc.exe –std=c99 -Wall -Werror -o args.exe args.c
```

Здесь запускается программа gcc.exe, которая из командной строки получает шесть аргументов: -std=c99, -Wall, -Werror, -o, args.exe и args.c.

Если программа написана на языке Си, то после ее запуска управление передается в функцию main, которая и получает аргументы командной строки:

```
int main(int argc, char** argv);
```

Данный вариант функции main получает два параметра:

- целое число (argc), обозначающее количество аргументов (элементов, разделенных пробелами) в командной строке при вызове (следует иметь в виду, что само имя программы также учитывается),
- указатель на массив строк (argv), где каждая строка - это отдельный аргумент из командной строки.

Для приведенного выше примера значение argc равно 7, а массив строк argv определяется как {«gcc.exe», «-std=c99», «-Wall», «-Werror», «-o», «args.exe», «args.c», NULL}.

То, что в программу передаются данные, не означает, что функция main должна их обрабатывать. Если функция main определена без параметров (`int main(void)`), то получить доступ к аргументам командной строки невозможно, но при этом вы можете указывать их при запуске.

Изучите вывод программы, приведенной ниже, запуская ее с различными аргументами командной строки.

```
#include <stdio.h>

int main(int argc, char **argv)
{
    printf("argc = %d\n", argc);

    for (int i = 0; i < argc; i++)
        puts(argv[i]);

    return 0;
}
```

Имена argc и argv не являются обязательными (т.е. вы может использовать любые), но лучше придерживаться именно этих имен, чтобы ваши программы были более понятны не только вам, но и другим программистам.

Чаще всего в программу при запуске передаются имена файлов и «опции» (ключи), которые влияют на процесс выполнения программы.

Пример 3.

Программа получает на вход текстовый файл, имя которого передается через аргументы командной строки. Текстовый файл содержит последовательность целых чисел. Необходимо вывести эту последовательность на экран.

```
#include <stdio.h>

void print_nums(FILE *f)
{
    int num;

    while (fscanf(f, "%d", &num) == 1)
        printf("%d\n", num);
}

int main(int argc, char** argv)
{
    FILE *f;

    if (argc != 2)
    {
        fprintf(stderr, "num_reader.exe <file-name>\n");

        return -1;
    }

    f = fopen(argv[1], "r");
    if (f == NULL)
    {
        fprintf(stderr, "I/O error\n");

        return -2;
    }
```

```
    print_nums(f);

    fclose(f);

    return 0;
}
```

5. Обработка ошибочных ситуаций

Информирование об ошибках

Большинство функций стандартной библиотеки при возникновении ошибки возвращают отрицательное число или NULL и записывают в глобальную переменную errno код ошибки. Эта переменная определена в заголовочном файле errno.h. Заметим, что в случае успешного выполнения функции эта переменная просто не изменяется и может содержать любой мусор, поэтому проверять ее имеет смысл лишь в случае, если ошибка действительно произошла.

При использовании функций, устанавливающих errno, можно сверить значение errno со значениями ошибок, определенных в include-файле <errno.h>, или же использовать функции perror и strerror. Если нужно распечатать сообщение о стандартной ошибке - используется perror; если сообщение об ошибке нужно расположить в строке, то используется strerror.

perror (параграф 7.19.10.4 стандарта)

```
#include <stdio.h>

void perror(const char *message);
```

Функция perror преобразует значение глобальной переменной errno в строку и записывает эту строку в stderr. Если значение параметра message не равно нулю, то сначала записывается сама строка, за ней ставится двоеточие, а затем следует сообщение об ошибке, определяемое конкретной реализацией.

strerror (параграф 7.21.6.2 стандарта)

```
#include <string.h>

char* strerror(int errnum);
```

Функция strerror возвращает указатель на строку, содержащую системное сообщение об ошибке, связанной со значением errnum. Эту строку не следует менять ни при каких обстоятельствах.

Обработка ошибок выделения ресурсов

Рассмотрим следующую задачу. На вход программе подается текстовый файл, который содержит последовательность из целых чисел. Необходимо переписать числа больше заданного в другой файл. Имена файлов и число передаются как параметры командной строки.

В программе, которая приведена в первом решении, используется подход обычно используемый студентами. Он характеризуется использованием большого количества операторов return (когда в программе возникает ошибочная ситуация, программа тут же завершается). Использование большого количества операторов return приводит к многочисленному закрытию файлов. Как правило, часть этих закрытий забывается.

Решение 1.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

void usage(void)
{
    printf("example.exe <source file> <destination file> <value>\n");
}

void select(FILE* fsrc, FILE* fdst, int val)
{
    int num;

    while (fscanf(fsrc, "%d", &num) == 1)
        if (num > val)
            fprintf(fdst, "%d\n", num);
}

int main(int argc, char** argv)
{
    FILE *fsrc, *fdst;
    int val;
    char *end_ptr;

    if(argc != 4)
    {
        usage();

        return -1;
    }

    fsrc = fopen(argv[1], "r");
    if (!fsrc)
    {
        fprintf(stderr, "Could not open %s because of %s\n",
                                            argv[1], strerror(errno));
        return -2;
    }

    fdst = fopen(argv[2], "w");
    if (!fdst)
    {
        fprintf(stderr, "Could not create %s because of %s\n",
                                            argv[2], strerror(errno));
        fclose(fsrc);

        return -3;
    }

    val = strtol(argv[3], &end_ptr, 10);
    if (*end_ptr)
    {
        fprintf(stderr, "Could not conver string to number (%s)\n",
                                                        strerror(errno));
        fclose(fsrc);
        fclose(fdst);

        return -4;
```

```
    }

    select(fsrc, fdst, val);

    fclose(fdst);
    fclose(fsrc);

    return 0;
}
```

В программе из решения 2 используется структурный подход: у каждой подпрограммы должна быть одна точка входа и одна точка выхода. Благодаря этому, отпадает необходимость закрывать файл несколько раз. Это нужно сделать только один раз, в случае его успешного открытия. Недостатком подобного подхода является зависимость вложенности операторов от числа ресурсов.

Решение 2.

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

void usage(void)
{
    printf("example.exe <source file> <destination file> <value>\n");
}

void select(FILE* fsrc, FILE* fdst, int val)
{
    int num;

    while (fscanf(fsrc, "%d", &num) == 1)
        if (num > val)
            fprintf(fdst, "%d\n", num);
}

int main(int argc, char** argv)
{
    FILE *fsrc, *fdst;
    int val;
    char *end_ptr;
    int rc = 0;

    if (argc != 4)
    {
        usage();
        rc = -1;
    }
    else
    {
        fsrc = fopen(argv[1], "r");
        if (fsrc)
        {
            fdst = fopen(argv[2], "w");
            if (fdst)
            {
                val = strtol(argv[3], &end_ptr, 10);
                if (!(*end_ptr))
                    select(fsrc, fdst, val);
                else
                {
                    fprintf(stderr, "Could not conver string to number (%s)\n",
                                                                strerror(errno));
                    rc = -4;
                }
```

```
                fclose(fdst);
            }
            else
            {
                fprintf(stderr, "Could not create %s because of %s\n",
                                                        argv[2], strerror(errno));
                rc = -3;
            }
            fclose(fsrc);
        }
        else
        {
            fprintf(stderr, "Could not open %s because of %s\n",
                                                argv[1], strerror(errno));
            rc = -2;
        }
    }

    return rc;
}
```

В программе из решения 3 для исправления недостатка структурного подхода используется оператор goto [2 и ссылки в статье]. В данном случае этот оператор используется для организации одной точки выхода и передает управление всегда в одном направлении. Подобный подход часто используется в ядре операционной системы Linux и драйверах этой ОС. Его использование требует дисциплины проставления меток.

Решение 3

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <errno.h>

void usage(void)
{
    printf("example.exe <source file> <destination file> <value>\n");
}

void select(FILE* fsrc, FILE* fdst, int val)
{
    int num;

    while (fscanf(fsrc, "%d", &num) == 1)
        if (num > val)
            fprintf(fdst, "%d\n", num);
}

int main(int argc, char** argv)
{
    FILE *fsrc, *fdst;
    int val;
    char *end_ptr;
    int rc = 0;

    if (argc != 4)
    {
        usage();
        rc = -1;
        goto fin;
    }

    fsrc = fopen(argv[1], "r");
    if (!fsrc)
    {
        fprintf(stderr, "Could not open %s because of %s\n", argv[1], strerror(errno));
```

```
        rc = -2;
        goto fin;
    }

    fdst = fopen(argv[2], "w");
    if (!fdst)
    {
        fprintf(stderr, "Could not create %s because of %s\n", argv[2], strerror(errno));
        rc = -3;
        goto close_src;
    }

    val = strtol(argv[3], &end_ptr, 10);
    if (*end_ptr)
    {
        fprintf(stderr, "Could not conver string to number (%s)\n", strerror(errno));
        rc = -4;
        goto close_all;
    }

    select(fsrc, fdst, val);

close_all:
    fclose(fdst);

close_src:
    fclose(fsrc);

fin:
    return rc;
}
```

При реализации задач, требующих работы с ресурсами, необходимо использовать структурный подход (т.е. подход 2). Подход 3 (оператор goto) можно использовать только в описанном применении и только при умении самостоятельно обосновать использование этого подхода без ссылок на лектора или преподавателя, проводящего лабораторные работы. В качестве отправной точки для такого обоснования (кроме [2]) рекомендуется познакомиться со статьями [3, 4].

6. Двоичные файлы

Текстовые файлы хранят информацию в виде понятном для человека. Можно хранить данные непосредственно в двоичном виде (в таком представлении, в котором они хранятся в оперативной памяти). Для этих целей используются двоичные файлы.

В отличие от текстового файла двоичный файл структура данных с произвольным доступом: можно установить внутренний указатель файла на интересующую позицию и прочитать (записать) информацию именно туда.

fwrite (параграф 7.19.8.2 стандарта)

```
#include <stdio.h>

size_t fwrite(const void *ptr, size_t size, size_t count, FILE *f);
```

Функция fwrite записывает в файл, связанный с файловой переменной f, данные из буфера ptr. Количество элементов в буфере ptr указывается в переменной count, а размер каждого элемента – в переменной size (размер указывается в байтах).

Замечание.

Вместо буфера можно передать адрес любой переменной.

Функция fwrite возвращает количество удачно записанных элементов. Если это количество не совпадает со значением count, то произошла ошибка.

fread (параграф 7.19.8.1 стандарта)

```
#include <stdio.h>

size_t fread(void *ptr, size_t size, size_t count, FILE *f);
```

Функция fread считывает из файла, связанного с файловой переменной f, данные и помещает их в буфер ptr. Количество считываемых элементов буфера указывается в переменной count, а размер каждого элемента – в переменной size (размер указывается в байтах).

Функция возвращает число удачно прочитанных элементов. Если возвращаемое значение отличается от количества элементов, значит произошла ошибка или был достигнут конец файла. Вы можете использовать функции ferror или feof для определения проблемы — произошла ошибка или был достигнут конец файла.

fseek (параграф 7.19.9.2 стандарта)

```
#include <stdio.h>

int fseek(FILE *f, long int offset, int origin);
```

Функция fseek устанавливает внутренний указатель положения в файле, связанном с файловой переменной f, в новую позицию offser относительно origin. origin может принимать три значения:

SEEK_SET начало файла;
SEEK_CUR текущее положение файла;
SEEK_END конец файла.

В случае успеха, функция возвращает нулевое значение.

ftell (параграф 7.9.19.4 стандарта)

```
#include <stdio.h>

long int ftell(FILE *f);
```

Функция ftell возвращает текущее положение внутреннего указателя в файле, связанном с файловой переменной f.

Для двоичных файлов возвращается значение соответствующее количеству байт от начала файла. Для текстовых файлов это значение может не соответствовать точному количеству байт от начала файла.

Замечание

Имеет смысл так же обратить внимание на функции fsetpos (7.19.9.3) и fgetpos (7.19.9.1), которые возвращают признак успешности выполнения запрошенной операции.

Создание двоичного файла.

```
#include <stdio.h>

int main(void)
{
    FILE *f;
    int number;
    size_t wrote;

    setbuf(stdout, NULL);

    f = fopen("data.bin", "wb");
    if (!f)
    {
        printf("Open file error\n");

        return 1;
    }

    printf("Input an integer: ");
    if (scanf("%d", &number) == 1)
    {
        wrote = fwrite(&number, sizeof(int), 1, f);

        printf("fwrite: %s\n", wrote == 1 ? "OK" : "ERROR");
    }

    fclose(f);

    return 0;
}
```

Выполните программу и посмотрите содержимое файла data.bin (в MSYS2 это можно сделать, например, с помощью следующей команды «hexdump -C data.bin»).

Чтение данных из двоичного файла.

```
#include <stdio.h>

int main(void)
{
    FILE *f;
    int number;
    size_t read;

    setbuf(stdout, NULL);

    f = fopen("data.bin", "rb");
    if (!f)
    {
        printf("Open file error\n");

        return 1;
    }
```

```
    read = fread(&number, sizeof(int), 1, f);

    printf("fread: %s (%d)\n", read == 1 ? "OK" : "ERROR", number);

    fclose(f);

    return 0;
}
```

Управление позицией чтения/записи в файле.

```
#include <stdio.h>

int main(void)
{
    FILE *f;
    int number;
    long int pos;
    size_t read;
    int rc;

    setbuf(stdout, NULL);

    f = fopen("data.bin", "rb");
    if (!f)
    {
        printf("Open file error\n");

        return 1;
    }

    pos = ftell(f);
    printf("ftell: pos = %ld\n", pos);

    read = fread(&number, sizeof(int), 1, f);
    printf("fread: %s (%d)\n", read == 1 ? "OK" : "ERROR", number);

    pos = ftell(f);
    printf("ftell: pos = %ld\n", pos);

    rc = fseek(f, 0, SEEK_SET);
    printf("fseek: %s\n", rc == 0 ? "OK" : "ERROR");

    pos = ftell(f);
    printf("ftell: pos %ld\n", pos);

    read = fread(&number, sizeof(int), 1, f);
    printf("fread: %s (%d)\n", read == 1 ? "OK" : "ERROR", number);

    fclose(f);

    return 0;
}
```

7. Литература

1. http://younglinux.info/sites/default/files/programmingC.pdf
2. http://eli.thegreenplace.net/2009/04/27/using-goto-for-error-handling-in-c
3. https://habrahabr.ru/post/271131/

4. https://habrahabr.ru/post/303712/